C++ is a general purpose, high level, compiler based object oriented programming language.

By using C++ different types of software can be designed.

Ex:

OS: DOS, Windows, Unix…

Editor: Notepad, Wordpad, Edit plus…

Commercial apps: Billing system in hotels, restaurants, super markets etc.,

Databases: Oracle, SQL server, My sql, mongo db etc.,

Translators:

1. Compiler and interpreter are used to convert high level program to machine code
2. Assembler is used to convert low level programs to machine code

Device drivers: A program that tells the OS how a device works.

PC and mobile games: Games are designed in c++. Ex: Snake game in mobiles – nokia

Protocols:

A protocol is a set of rules to be followed by two or more communicating bodies (computers) with each other.

ftp-file transfer protocol

dsp

http

tcp/ip

CDMA- code division Mulitple Access

We can design protocols using c++. FTP, HTTP are designed in c++. 90% of telecom protocols.

An Annotated Hello World  
The following dead simple C++ program shows all of the components of a C++ program:

// helloworld.cpp  
#include <iostream> // C++ standard headers don't use ".h"  
/\* main() in C++ takes argc (the number of arguments) and  
argv (an array of arguments) \*/  
int main(int argc, char\*\* argv)  
{  
// "std" is the standard C++ namespace, like a Java package  
std::cout << "Hello, World!" << std::endl;  
return 0;  
}

**Namespaces**Namespaces, like packages in Java, address the problem of naming conflicts between  
different pieces of code. For example, you might be writing some code that has a function  
called foo(). One day, you decide to start using a third-party library, which also has a  
foo() function. The compiler has no way of knowing which version of foo() you are  
referring to within your code.  
Namespaces solve this problem by allowing you to define the context in which names are  
defined. To place code in a namespace, simply enclose it within a namespace block:  
// namespaces.h  
namespace mycode {  
void foo();  
}

2  
// namespaces.cpp  
#include <iostream>  
namespace mycode {  
void foo() {  
std::cout << "foo() in the mycode namespace" << std::endl;  
}  
}  
To call the namespace-enabled version of foo():  
mycode::foo(); // calls the "foo" function in the "mycode" namespace  
To avoid being explicit about the namespace with every call, use using:  
#include "namespaces.h"  
using namespace mycode;  
int main(int argc, char\*\* argv)  
{  
foo(); // implies mycode::foo();  
}  
Namespaces are a great example of the difference between a hacked together C++  
program and a production quality program. The programs you write should be  
namespace-savvy.  
Preprocessor Directives  
Java programmers may not be familiar with the notion of a preprocessor. A preprocessor  
is like a compiler that runs over the code before the real compiler does its work. In C and  
C++, lines that begin with # contain commands for the preprocessor. In C, the  
preprocessor is often used to create faux constants and gross inlined functions known as  
macros. These preprocessor features are present in C++ but are rarely used.  
The main use of the preprocessor in C++ is the #include mechanism. Unlike Java, C and  
C++ functions are declared separately from their definitions. The declaration (which for  
C++ classes is confusingly referred to as a class definition) is usually placed in a file that  
ends in .h. Typically, C++ header files also make use of the preprocessor to make sure

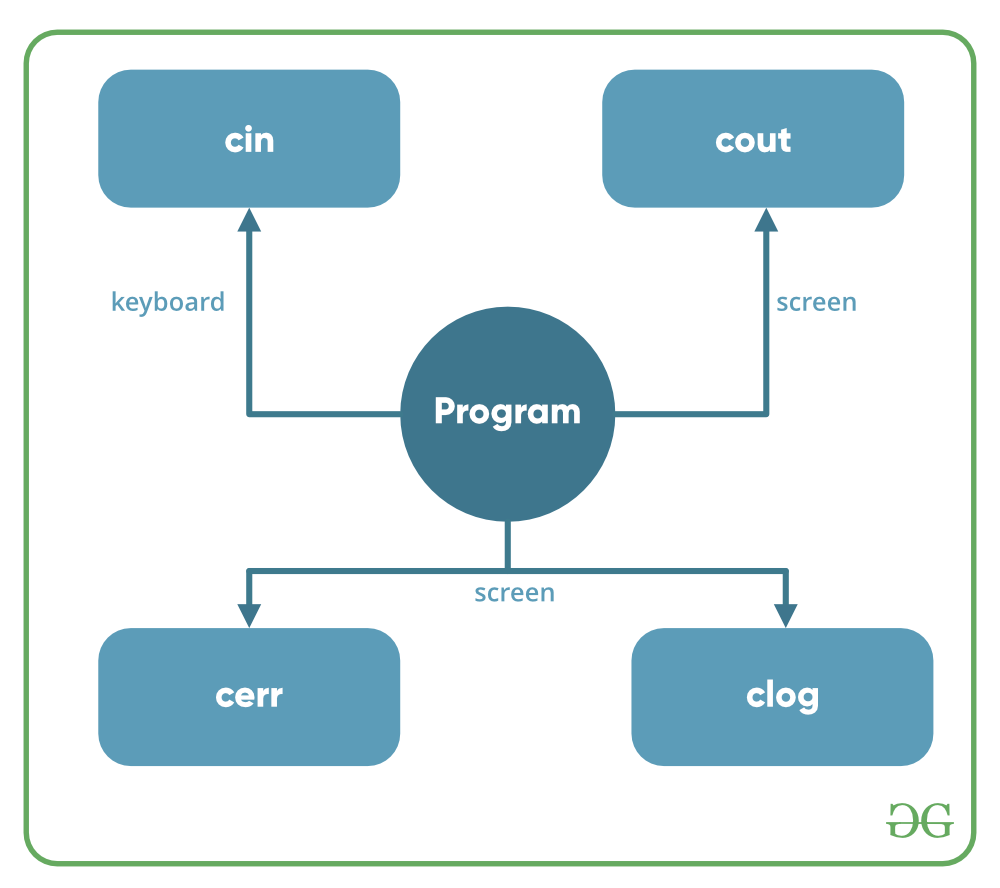
3  
that they are only included a single time. Each header defines a unique symbol and will  
skip its content if the symbol is already defined:  
// myheader.h  
#ifndef \_\_MYHEADER\_H\_\_  
#define \_\_MYHEADER\_H\_\_  
// content of the header goes here  
#endif // \_\_MYHEADER\_H\_\_  
Worth noting: In C++, the standard language headers are included using angle brackets  
(< and >) and do not traditionally use ".h", as shown above with <iostream>. Why not?  
Well, all of the standard C headers are also available within C++. When using the C  
headers, you do include the .h.  
Variables, Types, Conditionals, Loops, etc.  
Most of the basic types and other constructs are similar between C, Java, and C++. For  
example, the following snippet of code works in all three languages:  
int result = inValue;  
for (int i = inValue - 1; i > 1; i--) {  
result \*= i;  
}  
return result;  
Dynamically Created Arrays and Heap Memory  
Recall that standard arrays live on the stack and their size is determined at compile time:  
int\* myArray[30];  
To create an array whose size is determined at runtime, you declare it on the heap by  
allocating new memory:  
int\* myArray = new int[arraySize];  
Heap memory must be released manually by calling delete. When releasing memory that  
was allocated an array, you must use the bracket version of delete:

4  
delete[] myArray;  
Heap memory is also used in C to achieve pass-by-reference. We will get into pointer  
mechanics later in the quarter. For now, we don't need to use pointers for pass-by-  
reference because we have...  
References  
A reference is basically a short-hand for a pointer without the messiness of dereferencing  
or the ambiguity of ownership. Java doesn't have an analogous concept because object  
arguments to Java methods are passed by reference automatically. Since Java has an  
object class for every corresponding basic type (e.g. Integer for int), there is always a  
simple way to achieve pass-by-reference.  
In C++, the & character is used to indicate that a variable is a reference. For now, we'll  
just focus on references as parameters to functions and methods. There are some  
subtleties we'll get into later.  
The first version of addOne() below does not use a reference, so the variable that is  
passed in remains unchanged. The second version uses a reference, so the underlying  
variable is actually modified within the function. References can be used for basic types  
as well as more complex types, like classes.  
void addOne(int i)  
{  
i++; // has no real effect since this is a copy of the original  
}  
void addOne(int& i)  
{  
i++; // actually changes the original variable  
}  
Strings  
C++ programmers make use of both traditional C-style strings (arrays of characters with  
a null terminator) and the C++ string class. The string class works much like the Java  
String class, which is to say that it behaves like you would expect:  
#include <string>  
#include <iostream>  
using namespace std;

5  
int main(int argc, char\*\* argv)  
{  
string str1 = "Hello";  
string str2 = "World";  
string str3 = str1 + " " + str2;  
cout << "str1 is " << str1 << endl;  
cout << "str2 is " << str2 << endl;  
cout << "str3 is " << str3 << endl;  
if (str3 == "Hello World") {  
cout << "str3 is what it should be" << endl;  
} else {  
cout << "hmmm... str3 isn't what it should be" << endl;  
}  
return 0;  
}  
Classes  
For people who don't have much experience with Object-Oriented Programming, we'll be  
talking more about classes in (no pun intended) an upcoming class. Until then, just think  
of classes as similar to C structs with associated functions. Here is the syntax for a class  
definition:  
// Calculator.h  
class Calculator  
{  
public: // external code can call these methods  
Calculator(); // constructor  
~Calculator(); // destructor  
int add(int num1, int num2); // method  
float divide(float numerator, float denominator); // method  
bool getAllowNegatives(); // method  
void setAllowNegatives(bool inValue); // method  
protected: // external code can't access these members  
bool fAllowNegatives; // data member

6  
}; // note the semicolon at the end!  
The functionality, or methods, of the classes are defined as shown:  
// Calculator.cpp  
#include <iostream>  
#include "Calculator.h"  
Calculator::Calculator()  
{  
fAllowNegatives = false; // initialize the data member  
}  
Calculator::~Calculator()  
{  
// nothing much to do in terms of cleanup  
}  
int Calculator::add(int num1, int num2)  
{  
if (!getAllowNegatives() && (num1 < 0 || num2 < 0)) {  
std::cout << "Illegal negative number passed to add()" <<  
std::endl;  
return 0;  
}  
return num1 + num2;  
}  
float Calculator::divide(float numerator, float denominator)  
{  
if (!getAllowNegatives() && (numerator < 0 || denominator < 0)) {  
std::cout << "Illegal negative number passed to divide()" <<  
std::endl;  
return 0;  
}  
return (numerator / denominator);  
}  
bool Calculator::getAllowNegatives()

7  
{  
return fAllowNegatives;  
}  
void Calculator::setAllowNegatives(bool inValue)  
{  
fAllowNegatives = inValue;  
}  
Finally, here's how other parts of the code use and interact with the class:  
// CalculatorTest.cpp  
#include <iostream>  
#include “Calculator.h”  
using namespace std;  
int main(int argc, char\*\* argv)  
{  
Calculator myCalc; // stack-based Calculator  
myCalc.setAllowNegatives(true);  
int result = myCalc.add(2, 2);  
cout << “According to the calculator, 2 + 2 = “ << result << endl;  
Calculator\* myCalc2; // heap-based Calculator  
myCalc2 = new Calculator(); // allocate a new object  
myCalc2->setAllowNegatives(false);  
float result2 = myCalc2->divide(2.5, 0.5);  
cout << "According to the calculator, 2.5 / 0.5 = " << result2 <<  
endl;  
return 0;  
}  
If You Need More Review...  
For most of you, this hand-out was probably just a refresher. However, if you're coming  
from a C background or you need a little more help in any of these areas, check out  
Chapter 1 of the book. It has the same sections with similar examples, but much more  
detail



**Header files available in C++ for Input/Output operations are:**

1. **iostream**: iostream stands for standard input-output stream. This header file contains definitions of objects like cin, cout, cerr, etc.
2. **iomanip**: iomanip stands for input-output manipulators. The methods declared in these files are used for manipulating streams. This file contains definitions of setw, setprecision, etc.
3. **fstream**: This header file mainly describes the file stream. This header file is used to handle the data being read from a file as input or data being written into the file as output.

The two keywords **cout in C++** and **cin in C++** are used very often for printing outputs and taking inputs respectively. These two are the most basic methods of taking input and printing output in C++. To use cin and cout in C++ one must include the header file *iostream* in the program.

insertion operator(**<<**)

extraction operator(**>>**)

* **Un-buffered standard error stream (cerr)**: The C++ cerr is the standard error stream that is used to output the errors. This is also an instance of the ostream class. As cerr in C++ is un-buffered so it is used when one needs to display the error message immediately. It does not have any buffer to store the error message and display it later.
* The main difference between cerr and cout comes when you would like to redirect output using “cout” that gets redirected to file if you use “cerr” the error doesn’t get stored in file.(This is what un-buffered means ..It can’t store the message)

**Type Casting**

Converting an expression of a given type into another type is known as *type-casting*. We have already seen some ways to type cast:

**Implicit conversion**

Implicit conversions do not require any operator. They are automatically performed when a value is copied to a compatible type. For example:

|  |  |  |
| --- | --- | --- |
| 1 2 3 | *short* a=2000;  *int* b;  b=a; |  |

Here, the value of a has been promoted from short to int and we have not had to specify any type-casting operator. This is known as a standard conversion. Standard conversions affect fundamental data types, and allow conversions such as the conversions between numerical types (short to int, int to float, double to int...), to or from bool, and some pointer conversions. Some of these conversions may imply a loss of precision, which the compiler can signal with a warning. This warning can be avoided with an explicit conversion.  
  
Implicit conversions also include constructor or operator conversions, which affect classes that include specific constructors or operator functions to perform conversions. For example:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 | *class* A {};  *class* B { *public*: B (A a) {} };  A a;  B b=a; |  |

Here, an implicit conversion happened between objects of class A and class B, because B has a constructor that takes an object of class A as parameter. Therefore implicit conversions from A to B are allowed.

**Explicit conversion**

C++ is a strong-typed language. Many conversions, specially those that imply a different interpretation of the value, require an explicit conversion. We have already seen two notations for explicit type conversion: functional and c-like casting:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 | *short* a=2000;  *int* b;  b = (*int*) a; *// c-like cast notation*  b = *int* (a); *// functional notation* |  |

The functionality of these explicit conversion operators is enough for most needs with fundamental data types. However, these operators can be applied indiscriminately on classes and pointers to classes, which can lead to code that while being syntactically correct can cause runtime errors. For example, the following code is syntactically correct:

|  |  |  |  |
| --- | --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 | *// class type-casting*  *#include <iostream>*  *using* *namespace* std;  *class* CDummy {  *float* i,j;  };  *class* CAddition {  *int* x,y;  *public*:  CAddition (*int* a, *int* b) { x=a; y=b; }  *int* result() { *return* x+y;}  };  *int* main () {  CDummy d;  CAddition \* padd;  padd = (CAddition\*) &d;  cout << padd->result();  *return* 0;  } |  |  |

The program declares a pointer to CAddition, but then it assigns to it a reference to an object of another incompatible type using explicit type-casting:

|  |  |  |
| --- | --- | --- |
|  | padd = (CAddition\*) &d; |  |

Traditional explicit type-casting allows to convert any pointer into any other pointer type, independently of the types they point to. The subsequent call to member result will produce either a run-time error or a unexpected result.  
  
In order to control these types of conversions between classes, we have four specific casting operators: dynamic\_cast, reinterpret\_cast, static\_cast and const\_cast. Their format is to follow the new type enclosed between angle-brackets (<>) and immediately after, the expression to be converted between parentheses.  
  
dynamic\_cast <new\_type> (expression)  
reinterpret\_cast <new\_type> (expression)  
static\_cast <new\_type> (expression)  
const\_cast <new\_type> (expression)  
  
The traditional type-casting equivalents to these expressions would be:  
  
(new\_type) expression  
new\_type (expression)  
  
but each one with its own special characteristics:

**dynamic\_cast**

dynamic\_cast can be used only with pointers and references to objects. Its purpose is to ensure that the result of the type conversion is a valid complete object of the requested class.  
  
Therefore, dynamic\_cast is always successful when we cast a class to one of its base classes:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 | *class* CBase { };  *class* CDerived: *public* CBase { };  CBase b; CBase\* pb;  CDerived d; CDerived\* pd;  pb = *dynamic\_cast*<CBase\*>(&d); *// ok: derived-to-base*  pd = *dynamic\_cast*<CDerived\*>(&b); *// wrong: base-to-derived* |  |

The second conversion in this piece of code would produce a compilation error since base-to-derived conversions are not allowed with dynamic\_cast unless the base class is polymorphic.  
  
When a class is polymorphic, dynamic\_cast performs a special checking during runtime to ensure that the expression yields a valid complete object of the requested class:

|  |  |  |  |
| --- | --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 | *// dynamic\_cast*  *#include <iostream>*  *#include <exception>*  *using* *namespace* std;  *class* CBase { *virtual* *void* dummy() {} };  *class* CDerived: *public* CBase { *int* a; };  *int* main () {  *try* {  CBase \* pba = *new* CDerived;  CBase \* pbb = *new* CBase;  CDerived \* pd;  pd = *dynamic\_cast*<CDerived\*>(pba);  *if* (pd==0) cout << "Null pointer on first type-cast" << endl;  pd = *dynamic\_cast*<CDerived\*>(pbb);  *if* (pd==0) cout << "Null pointer on second type-cast" << endl;  } *catch* (exception& e) {cout << "Exception: " << e.what();}  *return* 0;  } | Null pointer on second type-cast | [Edit & Run](https://www.cplusplus.com/doc/oldtutorial/typecasting/) |

|  |
| --- |
| **Compatibility note:** dynamic\_cast requires the Run-Time Type Information (RTTI) to keep track of dynamic types. Some compilers support this feature as an option which is disabled by default. This must be enabled for runtime type checking using dynamic\_cast to work properly. |

The code tries to perform two dynamic casts from pointer objects of type CBase\* (pba and pbb) to a pointer object of type CDerived\*, but only the first one is successful. Notice their respective initializations:

|  |  |  |
| --- | --- | --- |
| 1 2 | CBase \* pba = *new* CDerived;  CBase \* pbb = *new* CBase; |  |

Even though both are pointers of type CBase\*, pba points to an object of type CDerived, while pbb points to an object of type CBase. Thus, when their respective type-castings are performed using dynamic\_cast, pba is pointing to a full object of class CDerived, whereas pbb is pointing to an object of class CBase, which is an incomplete object of class CDerived.  
  
When dynamic\_cast cannot cast a pointer because it is not a complete object of the required class -as in the second conversion in the previous example- it returns a null pointer to indicate the failure. If dynamic\_cast is used to convert to a reference type and the conversion is not possible, an exception of type bad\_cast is thrown instead.  
  
dynamic\_cast can also cast null pointers even between pointers to unrelated classes, and can also cast pointers of any type to void pointers (void\*).

**static\_cast**

static\_cast can perform conversions between pointers to related classes, not only from the derived class to its base, but also from a base class to its derived. This ensures that at least the classes are compatible if the proper object is converted, but no safety check is performed during runtime to check if the object being converted is in fact a full object of the destination type. Therefore, it is up to the programmer to ensure that the conversion is safe. On the other side, the overhead of the type-safety checks of dynamic\_cast is avoided.

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 | *class* CBase {};  *class* CDerived: *public* CBase {};  CBase \* a = *new* CBase;  CDerived \* b = *static\_cast*<CDerived\*>(a); |  |

This would be valid, although b would point to an incomplete object of the class and could lead to runtime errors if dereferenced.  
  
static\_cast can also be used to perform any other non-pointer conversion that could also be performed implicitly, like for example standard conversion between fundamental types:

|  |  |  |
| --- | --- | --- |
| 1 2 | *double* d=3.14159265;  *int* i = *static\_cast*<*int*>(d); |  |

Or any conversion between classes with explicit constructors or operator functions as described in "implicit conversions" above.

**reinterpret\_cast**

reinterpret\_cast converts any pointer type to any other pointer type, even of unrelated classes. The operation result is a simple binary copy of the value from one pointer to the other. All pointer conversions are allowed: neither the content pointed nor the pointer type itself is checked.  
  
It can also cast pointers to or from integer types. The format in which this integer value represents a pointer is platform-specific. The only guarantee is that a pointer cast to an integer type large enough to fully contain it, is granted to be able to be cast back to a valid pointer.  
  
The conversions that can be performed by reinterpret\_cast but not by static\_cast are low-level operations, whose interpretation results in code which is generally system-specific, and thus non-portable. For example:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 | *class* A {};  *class* B {};  A \* a = *new* A;  B \* b = *reinterpret\_cast*<B\*>(a); |  |

This is valid C++ code, although it does not make much sense, since now we have a pointer that points to an object of an incompatible class, and thus dereferencing it is unsafe.

**const\_cast**

This type of casting manipulates the constness of an object, either to be set or to be removed. For example, in order to pass a const argument to a function that expects a non-constant parameter:

|  |  |  |  |
| --- | --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 | *// const\_cast*  *#include <iostream>*  *using* *namespace* std;  *void* print (*char* \* str)  {  cout << str << endl;  }  *int* main () {  *const* *char* \* c = "sample text";  print ( *const\_cast*<*char* \*> (c) );  *return* 0;  } | sample text | [Edit & Run](https://www.cplusplus.com/doc/oldtutorial/typecasting/) |

**typeid**

typeid allows to check the type of an expression:   
  
typeid (expression)  
  
This operator returns a reference to a constant object of type type\_info that is defined in the standard header file <typeinfo>. This returned value can be compared with another one using operators == and != or can serve to obtain a null-terminated character sequence representing the data type or class name by using its name() member.

|  |  |  |  |
| --- | --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 | *// typeid*  *#include <iostream>*  *#include <typeinfo>*  *using* *namespace* std;  *int* main () {  *int* \* a,b;  a=0; b=0;  *if* (*typeid*(a) != *typeid*(b))  {  cout << "a and b are of different types:\n";  cout << "a is: " << *typeid*(a).name() << '\n';  cout << "b is: " << *typeid*(b).name() << '\n';  }  *return* 0;  } | a and b are of different types:  a is: int \*  b is: int | [Edit & Run](https://www.cplusplus.com/doc/oldtutorial/typecasting/) |

When typeid is applied to classes typeid uses the RTTI to keep track of the type of dynamic objects. When typeid is applied to an expression whose type is a polymorphic class, the result is the type of the most derived complete object:

|  |  |  |  |
| --- | --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 | *// typeid, polymorphic class*  *#include <iostream>*  *#include <typeinfo>*  *#include <exception>*  *using* *namespace* std;  *class* CBase { *virtual* *void* f(){} };  *class* CDerived : *public* CBase {};  *int* main () {  *try* {  CBase\* a = *new* CBase;  CBase\* b = *new* CDerived;  cout << "a is: " << *typeid*(a).name() << '\n';  cout << "b is: " << *typeid*(b).name() << '\n';  cout << "\*a is: " << *typeid*(\*a).name() << '\n';  cout << "\*b is: " << *typeid*(\*b).name() << '\n';  } *catch* (exception& e) { cout << "Exception: " << e.what() << endl; }  *return* 0;  } | a is: class CBase \*  b is: class CBase \*  \*a is: class CBase  \*b is: class CDerived | [Edit & Run](https://www.cplusplus.com/doc/oldtutorial/typecasting/) |

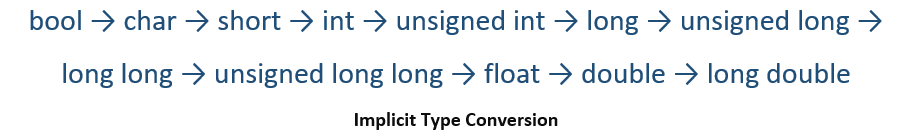
*Note: The string returned by member name of* [*type\_info*](https://www.cplusplus.com/type_info) *depends on the specific implementation of your compiler and library. It is not necessarily a simple string with its typical type name, like in the compiler used to produce this output.*   
Notice how the type that typeid considers for pointers is the pointer type itself (both a and b are of type class CBase \*). However, when typeid is applied to objects (like \*a and \*b) typeid yields their dynamic type (i.e. the type of their most derived complete object).  
  
If the type typeid evaluates is a pointer preceded by the dereference operator (\*), and this pointer has a null value, typeid throws a bad\_typeid exception.  
  
The compiler in the examples above generates names with [type\_info::name](https://www.cplusplus.com/type_info::name) that are easily readable by users, but this is not a requirement: a compiler may just return any string.

The type casting is a method of converting the value of one data type to another data type. It is also known as type conversion. In C++, there are two kinds of conversions, which are given below:

* Implicit Type Casting
* Explicit Type Casting

**Implicit Type Casting**

The implicit type casting happens **automatically** when converting a smaller data types to larger data types. The compiler implicitly typecast the smaller data types to the larger data types. No data will be lost in this process.



**Example**

The example below shows how the implicit type casting is done in C++.

#include <iostream>

using namespace std;

int main (){

char num\_char = 65;

//implicit casting

short num\_short = num\_char;

int num\_int = num\_short;

long num\_long = num\_int;

float num\_float = num\_long;

double num\_double = num\_float;

long double num\_long\_double = num\_double;

//printing variables

cout<<"num\_char = "<<num\_char<<"\n";

cout<<"num\_short = "<<num\_short<<"\n";

cout<<"num\_int = "<<num\_int<<"\n";

cout<<"num\_long = "<<num\_long<<"\n";

cout<<"num\_float = "<<num\_float<<"\n";

cout<<"num\_double = "<<num\_double<<"\n";

cout<<"num\_long\_double = "<<num\_long\_double<<"\n";

}

The output of the above code will be:

num\_char = A

num\_short = 65

num\_int = 65

num\_long = 65

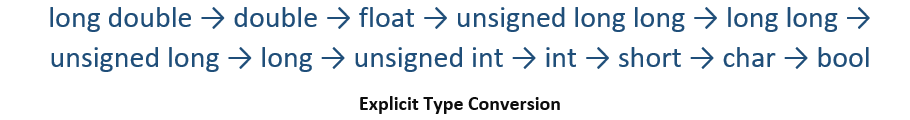
num\_float = 65

num\_double = 65

num\_long\_double = 65

**Explicit Type Casting**

The explicit type casting does not happen automatically. It is performed **manually** by calling the compiler explicitly to typecast the larger data types into the smaller data types. There might be a data loss in this process.



There are three major ways in which explicit conversion in C++ which are mentioned below:

1. C-style type casting
2. Function style type casting
3. Type conversion operators

**1. C-style type casting**

This is also known as **cast notation**. The syntax of this method is given below:

(new\_type)expression;

**2. Function style type casting**

The function style notation can also be used for type casting. The syntax of this style is given below:

new\_type(expression);

**Example**

The example below shows how the perform explicit type casting using **C-style** and **Function style** type casting.

#include <iostream>

using namespace std;

int main (){

long double num\_long\_double = 68.75;

//explicit casting - C style

double num\_double = (double) num\_long\_double;

float num\_float = (float) num\_double;

long num\_long = (long) num\_float;

//explicit casting - Function style

int num\_int = int(num\_long);

short num\_short = short(num\_int);

char num\_char = char(num\_short);

bool num\_bool = bool(num\_char);

//printing variables

cout<<boolalpha;

cout<<"num\_long\_double = "<<num\_long\_double<<"\n";

cout<<"num\_double = "<<num\_double<<"\n";

cout<<"num\_float = "<<num\_float<<"\n";

cout<<"num\_long = "<<num\_long<<"\n";

cout<<"num\_int = "<<num\_int<<"\n";

cout<<"num\_short = "<<num\_short<<"\n";

cout<<"num\_char = "<<num\_char<<"\n";

cout<<"num\_bool = "<<num\_bool<<"\n";

}

The output of the above code will be:

num\_long\_double = 68.75

num\_double = 68.75

num\_float = 68.75

num\_long = 68

num\_int = 68

num\_short = 68

num\_char = D

num\_bool = true

**3. Type conversion operators**

A Cast operator is an unary operator which forces one data type to be converted into another data type. C++ supports four types of casting:

1. static\_cast
2. dynamic\_cast
3. const\_cast
4. reinterpret\_cast

**Example**

The example below shows how to use type conversion operators.

#include <iostream>

using namespace std;

int main (){

float num\_float = 100.55;

//using static\_cast operator

int num\_int = static\_cast<int> (num\_float);

//printing variables

cout<<"num\_float = "<<num\_float<<"\n";

cout<<"num\_int = "<<num\_int<<"\n";

}

The output of the above code will be:

num\_float = 100.55

num\_int = 100

STL:

Vector:

C++ vector member functions can be put under the following categories:

construction, capacity, iterators, element access, and modifiers. Each of these categories has many functions. All these functions are not used in many applications. This article explains the most useful of these functions. With the exception of the modifiers category, not more than three functions for each category are explained in this article. The modifiers category can be broken down into more sub categories. In each of these sub categories, not more than three functions will be explained. If more than three functions are to be explained for a given category, then they will be taught by way of illustration.

## Construction/Destruction

The following code segments show different ways of creating the same vector:

vector <float> vtr;  
  
        vtr.push\_back(5.5);  
  
        vtr.push\_back(6.6);  
  
        vtr.push\_back(7.7);  
  
        vtr.push\_back(8.8);  
  
        vtr.push\_back(9.9);  
  
    vector <float> vtr(3);    //with initial number of elements  
  
        vtr[0] = 5.5;  
  
        vtr[1] = 6.6;  
  
        vtr[2] = 7.7;  
  
    vector <float> vtr(5, 0.0);    //No. Elements:5; each value:0.0  
  
    vector <float> vtr{5.5, 6.6, 7.7, 8.8, 9.9};    //initializing  
  
    vector <float> vtr = {5.5, 6.6, 7.7, 8.8, 9.9};    //constructing and copying  
  
    vector <float> vtr;  
  
    vtr = {5.5, 6.6, 7.7, 8.8, 9.9};  
  
    vector <float> vtr1{5.5, 6.6, 7.7, 8.8, 9.9};  
  
    vector <float> vtr2(vtr1);  
  
    const vector <float> vtr = {5.5, 6.6, 7.7, 8.8, 9.9};

A **const vector** is a vector whose elements cannot be changed. The values are read-only.

**Capacity**

**size() const noexcept**

The number of elements in a vector is returned by this member function. With the following code segment, the output is 5:

    vector <float> vtr = {5.5, 6.6, 7.7, 8.8, 9.9};  
  
  
    float sz = vtr.size();  
  
  
    cout << sz << '\n';  
  
  
empty() const noexcept

This method returns true (1) if the vector has no element and false (0) if the vector has at least one element. With the following code, the output is 1 (for true):

    vector <float> vtr = {};  
  
    bool bl = vtr.empty();  
  
    cout << bl << '\n';

**Iterator and Vector Access**

An iterator is an elaborated pointer. When the vector, **vtr** has been created, **vtr.begin()** would return an iterator, pointing to the first element of the list. It can then be incremented to access the elements after the first, accordingly.

When the vector, **vtr** has been created, **vtr.end()** would return an iterator, pointing just after the last element of the list. It can then be decremented to access the last element and elements before the last, accordingly. The following program illustrates this:

#include <iostream>  
  
#include <vector>  
  
using namespace std;  
  
int main()  
  
{  
  
    vector <float> vtr = {5.5, 6.6, 7.7, 8.8, 9.9};  
  
    vector<float>::iterator iterB = vtr.begin();  
  
    iterB++;  
  
    vector<float>::iterator iterE = vtr.end();  
  
    iterE--;  
  
    cout << \*iterB << ", " << \*iterE << ' ' << endl;  
  
    \*iterB = 66.66; \*iterE = 99.99;  
  
    cout << \*iterB << ", " << \*iterE << ' ' << endl;  
  
    return 0;  
  
}

The output is:

    6.6, 9.9  
  
    66.66, 99.99

The values of two elements were accessed, read and changed by two iterators.

**Element Access**

**at(i)**

This is similar to vtr[i], and it is better. It can be used to read or change the value of an element. Index counting begins from zero. The reader can test the following program:

#include <iostream>  
  
#include <vector>  
  
using namespace std;  
  
int main()  
  
{  
  
    vector <float> vtr = {5.5, 6.6, 7.7, 8.8, 9.9};  
  
    cout << vtr[1] << ", " << vtr[4] << ' ' << endl;  
  
    vtr[1] = 66.66; vtr[4] = 99.99;  
  
    cout << vtr[1] << ", " << vtr[4] << ' ' << endl;  
  
    return 0;  
  
}

The output is:

   6.6, 9.9  
  
    66.66, 99.99

The values of two elements were accessed, read and changed through referencing.

**Returning the First Value**

The following code returns (copies out) the first element:

    vector <float> vtr = {5.5, 6.6, 7.7, 8.8, 9.9};  
  
    float val = vtr.front();  
  
    cout << val << endl;

The output is, 5.5. The member function used here is: front().

**Returning the Last Value**

The following code returns (copies out) the last element:

    vector <float> vtr = {5.5, 6.6, 7.7, 8.8, 9.9};  
  
    float val = vtr.back();  
  
    cout << val << endl;

The output is, 9.9. The member function used here is: back()

# C++ Class Templates

Templates are powerful features of C++ which allows us to write generic programs. There are two ways we can implement templates:

* Function Templates
* Class Templates

# C++ Function Template

Templates are powerful features of C++ which allows us to write generic programs.

We can create a single function to work with different data types by using a template.

### Defining a Function Template

A function template starts with the keyword template followed by template parameter(s) inside <> which is followed by the function definition.

template <typename T>

T functionName(T parameter1, T parameter2, ...) {

// code

}

In the above code, T is a template argument that accepts different data types (int, float, etc.), and typename is a keyword.

When an argument of a data type is passed to functionName(), the compiler generates a new version of functionName() for the given data type.

### Calling a Function Template

Once we've declared and defined a function template, we can call it in other functions or templates (such as the main() function) with the following syntax

functionName<dataType>(parameter1, parameter2,...);

For example, let us consider a template that adds two numbers:

template <typename T>

T add(T num1, T num2) {

return (num1 + num2);

}

We can then call it in the main() function to add int and double numbers.

int main() {

int result1;

double result2;

// calling with int parameters

result1 = add<int>(2, 3);

cout << result1 << endl;

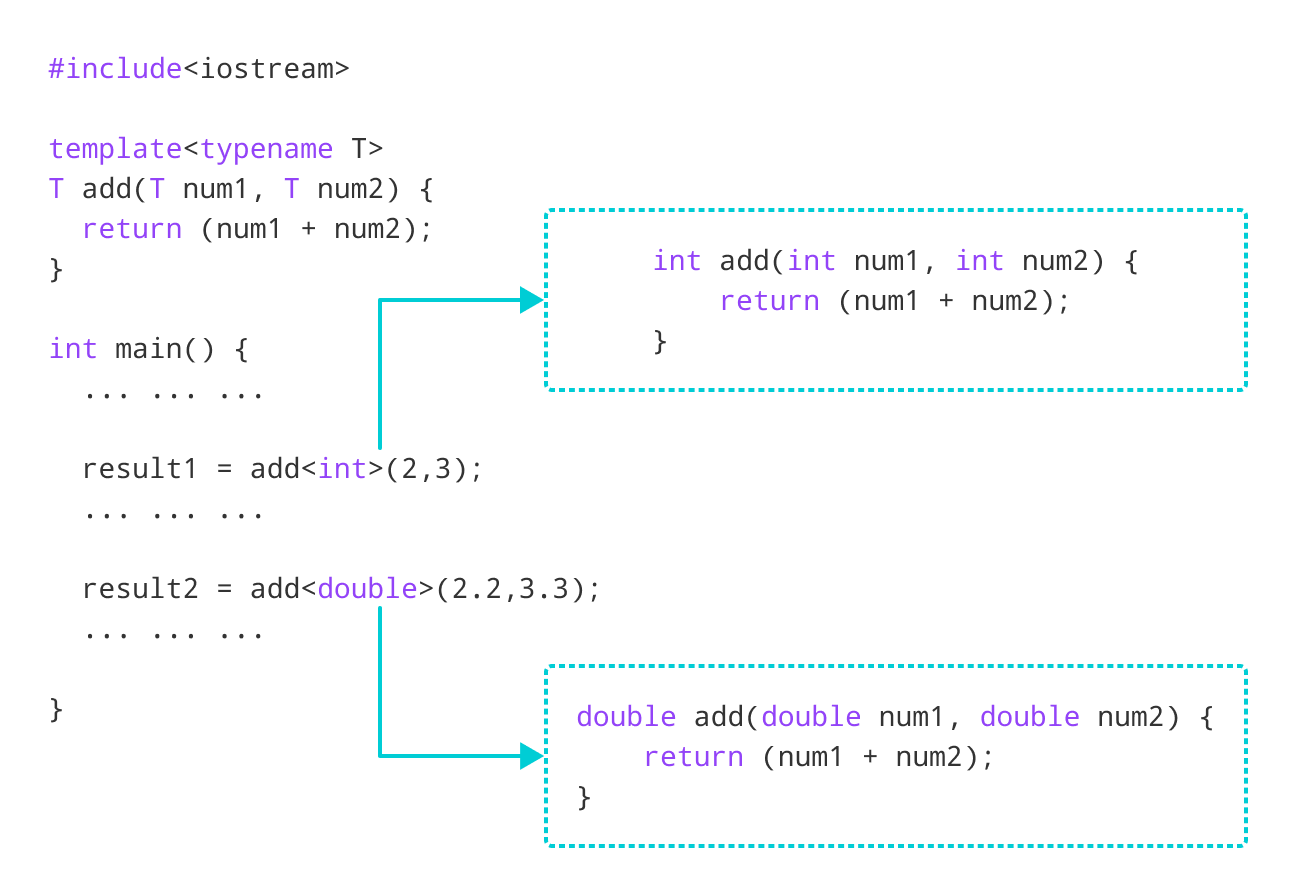
// calling with double parameters

result2 = add<double>(2.2, 3.3);

cout << result2 << endl;

return 0;

}

Function Call based on data types

### Example: Adding Two Numbers Using Function Templates

#include <iostream>

using namespace std;

template <typename T>

T add(T num1, T num2) {

return (num1 + num2);

}

int main() {

int result1;

double result2;

// calling with int parameters

result1 = add<int>(2, 3);

cout << "2 + 3 = " << result1 << endl;

// calling with double parameters

result2 = add<double>(2.2, 3.3);

cout << "2.2 + 3.3 = " << result2 << endl;

return 0;

}

**Output**

2 + 3 = 5

2.2 + 3.3 = 5.5

Similar to function templates, we can use class templates to create a single class to work with different data types.

Class templates come in handy as they can make our code shorter and more manageable.

## Class Template Declaration

A class template starts with the keyword template followed by template parameter(s) inside <> which is followed by the class declaration.

template <class T>

class className {

private:

T var;

... .. ...

public:

T functionName(T arg);

... .. ...

};

In the above declaration, T is the template argument which is a placeholder for the data type used, and class is a keyword.

Inside the class body, a member variable var and a member function functionName() are both of type T.

## Creating a Class Template Object

Once we've declared and defined a class template, we can create its objects in other classes or functions (such as the main() function) with the following syntax

className<dataType> classObject;

For example,

className<int> classObject;

className<float> classObject;

className<string> classObject;

## Example 1: C++ Class Templates

// C++ program to demonstrate the use of class templates

#include <iostream>

using namespace std;

// Class template

template <class T>

class Number {

private:

// Variable of type T

T num;

public:

Number(T n) : num(n) {} // constructor

T getNum() {

return num;

}

};

int main() {

// create object with int type

Number<int> numberInt(7);

// create object with double type

Number<double> numberDouble(7.7);

cout << "int Number = " << numberInt.getNum() << endl;

cout << "double Number = " << numberDouble.getNum() << endl;

return 0;

}

**Output**

int Number = 7

double Number = 7.7

In this program. we have created a class template Number with the code

template <class T>

class Number {

private:

T num;

public:

Number(T n) : num(n) {}

T getNum() { return num; }

};

Notice that the variable num, the constructor argument n, and the function getNum() are of type T, or have a return type T. That means that they can be of any type.

In main(), we have implemented the class template by creating its objects

Number<int> numberInt(7);

Number<double> numberDouble(7.7);

Notice the codes Number<int> and Number<double> in the code above.

This creates a class definition each for int and float, which are then used accordingly.

It is compulsory to specify the type when declaring objects of class templates. Otherwise, the compiler will produce an error.

//Error

Number numberInt(7);

Number numberDouble(7.7);

## Defining a Class Member Outside the Class Template

Suppose we need to define a function outside of the class template. We can do this with the following code:

template <class T>

class ClassName {

... .. ...

// Function prototype

returnType functionName();

};

// Function definition

template <class T>

returnType ClassName<T>::functionName() {

// code

}

Notice that the code template <class T> is repeated while defining the function outside of the class. This is necessary and is part of the syntax.

If we look at the code in **Example 1**, we have a function getNum() that is defined inside the class template Number.

We can define getNum() outside of Number with the following code:

template <class T>

class Number {

... .. ...

// Function prototype

T getnum();

};

// Function definition

template <class T>

T Number<T>::getNum() {

return num;

}

## Example 2: Simple Calculator Using Class Templates

This program uses a class template to perform addition, subtraction, multiplication and division of two variables num1 and num2.

The variables can be of any type, though we have only used int and float types in this example.

#include <iostream>

using namespace std;

template <class T>

class Calculator {

private:

T num1, num2;

public:

Calculator(T n1, T n2) {

num1 = n1;

num2 = n2;

}

void displayResult() {

cout << "Numbers: " << num1 << " and " << num2 << "." << endl;

cout << num1 << " + " << num2 << " = " << add() << endl;

cout << num1 << " - " << num2 << " = " << subtract() << endl;

cout << num1 << " \* " << num2 << " = " << multiply() << endl;

cout << num1 << " / " << num2 << " = " << divide() << endl;

}

T add() { return num1 + num2; }

T subtract() { return num1 - num2; }

T multiply() { return num1 \* num2; }

T divide() { return num1 / num2; }

};

int main() {

Calculator<int> intCalc(2, 1);

Calculator<float> floatCalc(2.4, 1.2);

cout << "Int results:" << endl;

intCalc.displayResult();

cout << endl

<< "Float results:" << endl;

floatCalc.displayResult();

return 0;

}

**Output**

Int results:

Numbers: 2 and 1.

2 + 1 = 3

2 - 1 = 1

2 \* 1 = 2

2 / 1 = 2

Float results:

Numbers: 2.4 and 1.2.

2.4 + 1.2 = 3.6

2.4 - 1.2 = 1.2

2.4 \* 1.2 = 2.88

2.4 / 1.2 = 2

In the above program, we have declared a class template Calculator.

The class contains two private members of type T: num1 & num2, and a constructor to initialize the members.

We also have add(), subtract(), multiply(), and divide() functions that have the return type T. We also have a void function displayResult() that prints out the results of the other functions.

In main(), we have created two objects of Calculator: one for int data type and another for float data type.

Calculator<int> intCalc(2, 1);

Calculator<float> floatCalc(2.4, 1.2);

This prompts the compiler to create two class definitions for the respective data types during compilation.

## C++ Class Templates With Multiple Parameters

In C++, we can use multiple template parameters and even use default arguments for those parameters. For example,

template <class T, class U, class V = int>

class ClassName {

private:

T member1;

U member2;

V member3;

... .. ...

public:

... .. ...

};

### Example 3: C++ Templates With Multiple Parameters

#include <iostream>

using namespace std;

// Class template with multiple and default parameters

template <class T, class U, class V = char>

class ClassTemplate {

private:

T var1;

U var2;

V var3;

public:

ClassTemplate(T v1, U v2, V v3) : var1(v1), var2(v2), var3(v3) {} // constructor

void printVar() {

cout << "var1 = " << var1 << endl;

cout << "var2 = " << var2 << endl;

cout << "var3 = " << var3 << endl;

}

};

int main() {

// create object with int, double and char types

ClassTemplate<int, double> obj1(7, 7.7, 'c');

cout << "obj1 values: " << endl;

obj1.printVar();

// create object with int, double and bool types

ClassTemplate<double, char, bool> obj2(8.8, 'a', false);

cout << "\nobj2 values: " << endl;

obj2.printVar();

return 0;

}

**Output**

obj1 values:

var1 = 7

var2 = 7.7

var3 = c

obj2 values:

var1 = 8.8

var2 = a

var3 = 0

In this program, we have created a class template, named ClassTemplate, with three parameters, with one of them being a default parameter.

template <class T, class U, class V = char>

class ClassTemplate {

// code

};

Notice the code class V = char. This means that V is a default parameter whose default type is char.

Inside ClassTemplate, we declare 3 variables var1, var2 and var3, each corresponding to one of the template parameters.

class ClassTemplate {

private:

T var1;

U var2;

V var3;

... .. ...

... .. ...

};

In main(), we create two objects of ClassTemplate with the code

// create object with int, double and char types

ClassTemplate<int, double> obj1(7, 7.7, 'c');

// create object with double, char and bool types

ClassTemplate<double, char, bool> obj2(8, 8.8, false);

Here,

|  |  |  |  |
| --- | --- | --- | --- |
| **Object** | **T** | **U** | **V** |
| obj1 | int | double | char |
| obj2 | double | char | bool |

For obj1, T = int, U = double and V = char.

For obj2, T = double, U = char and V = bool.

**Standard Containers**

A container is a holder object that stores a collection of other objects (its elements). They are implemented as class templates, which allows a great flexibility in the types supported as elements.  
  
The container manages the storage space for its elements and provides member functions to access them, either directly or through iterators (reference objects with similar properties to pointers).  
  
Containers replicate structures very commonly used in programming: dynamic arrays ([vector](https://www.cplusplus.com/vector)), queues ([queue](https://www.cplusplus.com/queue)), stacks ([stack](https://www.cplusplus.com/stack)), heaps ([priority\_queue](https://www.cplusplus.com/priority_queue)), linked lists ([list](https://www.cplusplus.com/list)), trees ([set](https://www.cplusplus.com/set)), associative arrays ([map](https://www.cplusplus.com/map))...  
  
Many containers have several member functions in common, and share functionalities. The decision of which type of container to use for a specific need does not generally depend only on the functionality offered by the container, but also on the efficiency of some of its members (complexity). This is especially true for sequence containers, which offer different trade-offs in complexity between inserting/removing elements and accessing them.  
  
[stack](https://www.cplusplus.com/stack), [queue](https://www.cplusplus.com/queue) and [priority\_queue](https://www.cplusplus.com/priority_queue) are implemented as *container adaptors*. Container adaptors are not full container classes, but classes that provide a specific interface relying on an object of one of the container classes (such as [deque](https://www.cplusplus.com/deque) or [list](https://www.cplusplus.com/list)) to handle the elements. The underlying container is encapsulated in such a way that its elements are accessed by the members of the *container adaptor* independently of the underlying *container* class used.

**Container class templates**

**Sequence containers**:

[**array**](https://www.cplusplus.com/reference/array/array/)

Array class (class template )

[**vector**](https://www.cplusplus.com/reference/vector/vector/)

Vector (class template )

[**deque**](https://www.cplusplus.com/reference/deque/deque/)

Double ended queue (class template )

[**forward\_list**](https://www.cplusplus.com/reference/forward_list/forward_list/)

Forward list (class template )

[**list**](https://www.cplusplus.com/reference/list/list/)

List (class template )

**Container adaptors**:

[**stack**](https://www.cplusplus.com/reference/stack/stack/)

LIFO stack (class template )

[**queue**](https://www.cplusplus.com/reference/queue/queue/)

FIFO queue (class template )

[**priority\_queue**](https://www.cplusplus.com/reference/queue/priority_queue/)

Priority queue (class template )

**Associative containers**:

[**set**](https://www.cplusplus.com/reference/set/set/)

Set (class template )

[**multiset**](https://www.cplusplus.com/reference/set/multiset/)

Multiple-key set (class template )

[**map**](https://www.cplusplus.com/reference/map/map/)

Map (class template )

[**multimap**](https://www.cplusplus.com/reference/map/multimap/)

Multiple-key map (class template )

**Unordered associative containers**:

[**unordered\_set**](https://www.cplusplus.com/reference/unordered_set/unordered_set/)

Unordered Set (class template )

[**unordered\_multiset**](https://www.cplusplus.com/reference/unordered_set/unordered_multiset/)

Unordered Multiset (class template )

[**unordered\_map**](https://www.cplusplus.com/reference/unordered_map/unordered_map/)

Unordered Map (class template )

[**unordered\_multimap**](https://www.cplusplus.com/reference/unordered_map/unordered_multimap/)

Unordered Multimap (class template )

# C++ Call by Reference: Using pointers [With Examples]

In this tutorial, we will learn about C++ call by reference to pass pointers as an argument to the function with the help of examples.

In the [C++ Functions](https://www.programiz.com/cpp-programming/function) tutorial, we learned about passing arguments to a function. This method used is called passing by value because the actual value is passed.

However, there is another way of passing arguments to a function where the actual values of arguments are not passed. Instead, the reference to values is passed.

For example,

// function that takes value as parameter

void func1(int numVal) {

// code

}

// function that takes reference as parameter

// notice the & before the parameter

void func2(int &numRef) {

// code

}

int main() {

int num = 5;

// pass by value

func1(num);

// pass by reference

func2(num);

return 0;

}

Notice the & in void func2(int &numRef). This denotes that we are using the address of the variable as our parameter.

So, when we call the func2() function in main() by passing the variable num as an argument, we are actually passing the address of num variable instead of the value **5**.

C++ Pass by Value vs. Pass by Reference

## Example 1: Passing by reference without pointers

#include <iostream>

using namespace std;

// function definition to swap values

void swap(int &n1, int &n2) {

int temp;

temp = n1;

n1 = n2;

n2 = temp;

}

int main()

{

// initialize variables

int a = 1, b = 2;

cout << "Before swapping" << endl;

cout << "a = " << a << endl;

cout << "b = " << b << endl;

// call function to swap numbers

swap(a, b);

cout << "\nAfter swapping" << endl;

cout << "a = " << a << endl;

cout << "b = " << b << endl;

return 0;

}

**Output**

Before swapping

a = 1

b = 2

After swapping

a = 2

b = 1

In this program, we passed the variables a and b to the swap() function. Notice the function definition,

void swap(int &n1, int &n2)

Here, we are using & to denote that the function will accept addresses as its parameters.

Hence, the compiler can identify that instead of actual values, the reference of the variables is passed to function parameters.

In the swap() function, the function parameters n1 and n2 are pointing to the same value as the variables a and b respectively. Hence the swapping takes place on actual value.

The same task can be done using the pointers. To learn about pointers, visit [C++ Pointers](https://www.programiz.com/cpp-programming/pointers).

## Example 2: Passing by reference using pointers

#include <iostream>

using namespace std;

// function prototype with pointer as parameters

void swap(int\*, int\*);

int main()

{

// initialize variables

int a = 1, b = 2;

cout << "Before swapping" << endl;

cout << "a = " << a << endl;

cout << "b = " << b << endl;

// call function by passing variable addresses

swap(&a, &b);

cout << "\nAfter swapping" << endl;

cout << "a = " << a << endl;

cout << "b = " << b << endl;

return 0;

}

// function definition to swap numbers

void swap(int\* n1, int\* n2) {

int temp;

temp = \*n1;

\*n1 = \*n2;

\*n2 = temp;

}

**Output**

Before swapping

a = 1

b = 2

After swapping

a = 2

b = 1

Here, we can see the output is the same as the previous example. Notice the line,

// &a is address of a

// &b is address of b

swap(&a, &b);

Here, the address of the variable is passed during the function call rather than the variable.

Since the address is passed instead of value, a dereference operator \* must be used to access the value stored in that address.

temp = \*n1;

\*n1 = \*n2;

\*n2 = temp;

\*n1 and \*n2 gives the value stored at address n1 and n2 respectively.

Since n1 and n2 contain the addresses of a and b, anything is done to \*n1 and \*n2 will change the actual values of a and b.

Hence, when we print the values of a and b in the main() function, the values are changed.